Probabilistic Inference Schemes for Sparsity Structures of Fortran 90 Array Intrinsics *

Rong–Guey Chang Jia–Shin Li Jenq Kuen Lee
Department of Computer Science,
National Tsing-Hua Univ., Hsinchu, Taiwan
{rgchang, jsli, jklee}@pllab.cs.nthu.edu.tw

Tyng–Ruey Chuang
Institute of Information Science,
Academia Sinica, Taipei, Taiwan
trc@iis.sinica.edu.tw

Abstract

Fortran 90 provides a rich set of array intrinsic functions, and they are useful to represent array expressions and data parallel programming. In the case that these intrinsic functions are applied to sparse data sets on distributed memory environments, it is however not supported by vendor Fortran 90 or HPF compilers. In a series of our research work, we attempt to provide parallel sparse supports for array intrinsics of Fortran 90[14]. In our design, we support sparse modules with overloaded array operations and intrinsics.

In this paper, we propose solutions to address the issues for the non-uniform distribution of sparse elements. Our work gives sparse inference schemes for the complete set of array operations and array intrinsics of Fortran 90 for non-uniform distributions of sparse datasets. Experiments are also done to observe our sampling techniques for sparse arrays with the Harwell-Boeing sparse matrix collection. We also demonstrate with our experiments and analytical model that a compiler can distinguish the performances among different OLAP programs with the help of the sparsity structures of the source arrays and our probabilistic inference schemes. Our experiments are done on IBM SP-2 and with the support of our sparse array intrinsics of Fortran 90.

1. Introduction

The usages of array intrinsic functions in Fortran 90 provide a rich source of parallelisms, and become more and more important due to the growing usages of Fortran 90, and the presence of the tutorial books such as *Numerical Recipes in Fortran 90*[11]. In the case that these intrinsic functions are applied to sparse data sets on distributed memory environments, it is however not supported by vendor Fortran 90 or HPF compilers. In a series of our research work, we attempt to provide parallel sparse supports for array intrinsics of Fortran 90[14].

*This work was supported in part by NSC of Taiwan under grant no. NSC-89-2218-E-007-023 and NSC-89-2219-E-007-012.

In our design, we support sparse modules with overloaded array operations and intrinsics. Sparse array functions are overloaded for array intrinsic interfaces so that programmers need not concern about the low-level details. Our library uses a two-level design. In the low-level routines, it requires the input sparse matrix to be specified with compression/distribution schemes for each function. Low-level communication routines use the Message Passing Interface (MPI) for communications of the sparse matrix among the processors. In the high-level representation, sparse array functions are overloaded for array intrinsic interfaces so that programmers need not concern about the low-level details. With our support, sparse matrix computation can now be expressed in Fortran 90 using high-level array expressions (just like dense matrix computation has always been), without concerning low-level coding of compression and distribution details. In addition, we also attempt to transform the high-level representations to low-level routines, automatically[14].

In this paper, we propose solutions to address the issues for the non-uniform distribution of sparse elements. Our work gives sparse inference schemes for the complete set of array operations and array intrinsics of Fortran 90 for non-uniform distributions of sparse datasets. In our work, we assume that the sparsity of source arrays can be profiled to obtain initial information. We then try to estimate the sparsity for the rest of the arrays (including temporary arrays) operated with sparse array intrinsics. We divide the problems into two categories, elementwise array operations and transformation array operations. We also give the theoretic results that the general problem related to sampling technique is NP-complete. Therefore, we use a heuristic algorithm to extract sub-blocks of arrays from sparse structures. Combining our previous research work on uniform sparse distributions[14] and this research work on non-uniform cases, we give sparse inference schemes for the complete set of array operations and array intrinsics of Fortran 90 with uniform or non-uniform distributions. Our work is particularly useful to give influences for a straight line of array operations with the knowledges of initial source arrays. Experiments are also done to observe our sampling techniques for sparse arrays with the Harwell-Boeing sparse matrix collection. We also demonstrate with our experiments and analytical model that a compiler can distinguish the performances among different OLAP programs with the help of the sparsity structures of the source arrays and our probabilistic inference schemes. Our experiments are done on IBM SP-2 and with the support of our sparse array intrinsics of Fortran 90.

The remainder of the paper is organized as follows. Section 2 gives the focus of this paper to present the methodologies for non-uniform cases. Finally, Section 3 gives our experimental results and Section 5 concludes this paper.

2. Inference Schemes with Non-Uniform Distributions

2.1. Dense Subarray Extraction/Sampling

In this section, we discuss the case for non-uniform cases. In our method, we first present a sampling technique to extract a set of blocks of sub-arrays with a given density from the non-uniform sparse arrays. Once the sub-block of arrays are gathered, they can be used for further inferences for the probability of sparse ratio of target arrays operated under F90 array intrinsics.

Let us consider the example shown in Figure 1. The sparsity of the sample array is with non-uniform distributions. Let us consider a sample array with non-uniform distributions. The question here is how to use blocks of sub-arrays to cover or to represent the sparsity of arrays. We feel this problem is very difficult at least for the following reasons. First, the sparsity structure of a dense subarray is usually irregular. What sparsity structures should be extracted so that the estimation of the sparsity will be precise? Second, once sparsity structures to be extracted decided, we also need to know the shape and sparsity ratio for the extractions. Finally, there might be several ways to extract subarrays. Figure 2 illustrates this issue. In this example, two different dense subarrays can be extracted with respect to the same input constraint, the shape constraint is 5×5 and the sparsity constraint is 0.5.

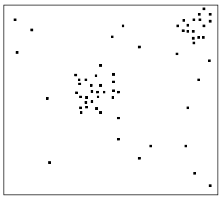

Figure 1. A sample sparse array with non-uniform distributions.

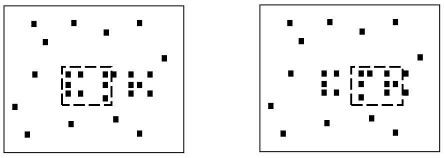

Figure 2. Two possible extracted dense subarrays in a sparse array.

In fact, the problem of extracting dense subarrays with a given density from sparse sub-arrays is NP-complete. We give our proof below. We would like to show that extracting dense subarrays in a sparse array is NP-complete. Let us define this problem as follows: For a given sparse array A, we want to extract dense subarrays in it to satisfy the specific shape constraint and sparsity constraint. Let $F = \{S_1, S_2, \cdots, S_k\}$ consist of all subarrays in A that satisfy the input constraints. Our aim is to find out whether it exists a subset $T \subseteq F$ of pairwise disjoint sets such that $\cup_{S_i \in T} S_i = \cup_{S_j \in F} S_j$. This problem is the exact cover problem that is a well-known NP-complete problem[5].

With this result in mind, we will explore further with heuristic algorithms. We devise algorithms to extract rectangular subarrays. The rectangular subarray has several good properties. First, it can be represented by block notations. Second, the rectangular arrays can be compressed and partitioned on distributed memory environments. Finally, the scheme can be generalized into higher-dimensional cases by adjusting input constraints.

Figure 3 shows our heuristic algorithm to extract dense subarrays in a sparse array. Initially, for a sparse array, the shape and sparsity constraints α and β for dense subarrays are given. Accordingly, the sparse array will be partitioned into many subarrays with re-

Input:

1. r-dimensional sparse array A.
2. The minimun shape of dense subarrays, $d_1, d_2, \cdots, d_r$ in each dimension, respectively.
3. The bound of sparsity ratio, α and β for dense subarrays.

Output: All possible dense subarrays in sparse array A.

Begin_Of_Algorithm

Step 1: Partition the input sparse array into `subarrays` according to the shape constraint.

Step 2: For each subarray do

Extract the `dense portion` that satisfies the following conditions with respect to two thresholds of sparsity ratio, α and β:

(1) the sparsity ratio greater than or equivalent to α, and

(2) the sparsity ratio of the region in boundary sections of each dimension must be greater than β, where $\alpha \geq \beta$.

End For

Step 3: Sort all subarrays by the sparsity ratio of dense subarrays.

Step 4: Select one subarray (say A_d) with highest density to do the followings.

Do-loop

We merge A_d with neighboring subarray. If the newly merged dense region (say A_{Nd}) does not satisfy the input sparsity constraint, then the merger for this subarray stops.

Otherwise, substitute A_{Nd} for A_d.

End do

End_Of_Algorithm

Figure 3. The algorithm for extracting dense subarrays.

spect to the shape constraint. In the step 2, the sparsity information of each subarray will be recorded to cut the regions that do not meet the two conditions α and β. The sub-block obtained in Step 1 will be cut if the sparsity ratio is less than α. A sub-block survives the cut for the test of the sparsity ratio of α will be further pruned to be smaller. The boundary section of a block will be pruned out if the sparsity of that section is smaller than β. Next, the remaining dense regions will be merged if the mergd block still satisfies the given sparsity constraint.

In the merging step, the index ranges must be conforming to each other when dense regions are merged. Figure 4 shows a two-dimensional case that two subarrays are merged after the partitioning phase. The dark regions in Figure 4 are the remaining dense regions. The grey region covering the smaller dark subarray at LHS is the one to be merged with the dense region at RHS. The index range of the grey region is conforming to the dense portion at the right-hand side. Therefore, we see the promoting of a smaller block into a bigger block for merging process. The merging order can be done by sorting sparsity ratios of neighboring dense subarrays. The merging process will be repeated until no neighboring dense subarrays can be merged with the newly merged subarray. After the merging process, the newly subarray must also meet the input constraints.

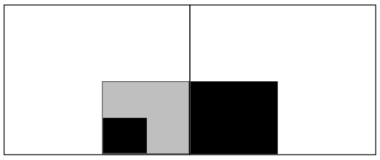

Figure 4. A scenario for merging regions in two neighboring subarrays

We give an example in the following to demonstrate how our algorithm works. Let us consider the sparse array shown in Figure 1 again. This shape of this sample array is 36×36 and the sparsity ratio is 0.045. Assume that the input shape and sparsity constraints are given as 6×6 and 0.3, respectively. In the first step, this sample array will be partitioned according to the input shape constraint. The result is shown Figure 5. Then the sparsity information of each subarray will be recorded. Let two thresholds α and β in Step 2 be 0.3 and 0.1. The dense regions of all subarrays will be extracted with respect to α, β, and the shape information. Note that the boundary sections of row dimension and column dimension will be stripped off if their sparsity ratios are less than β. The result after Step 2 is shown in Figure 6, where the grey regions are the regions extracted in Step 2. Next, dense neighboring subarrays are merged. The final results are shown in Figure 7.

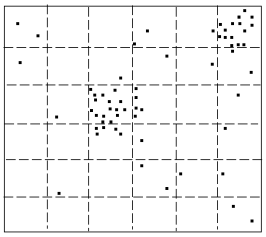

Figure 5. The sample sparse array after the partitioning phase.

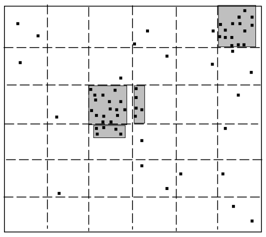

Figure 6. The intermediate result after the Step 3.

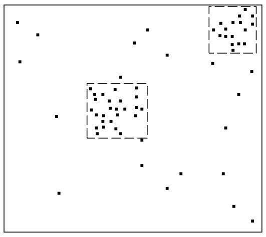

Figure 7. Two dense subarrays are extracted.

2.2. Segmented Inference Algorithm

In this section, we will present our segmented inference algorithm to estimate the sparsity ratio in sparse computation by utilizing the sparsity information provided by the extraction algorithm in Figure 3. Our algorithm is based on lattice calculus for calculating the regions involved. Let's first give the basic notations and operators used in our algorithm. we use three binary operators, $\cap$, $\cup$, and $-$, to describe the property that rectangular subarrays to be manipulated in sparse computation. Suppose that a sparse array A consists of two parts: a dense subarray A_1 and the remaining part A_r. Then the relationship between A_1 and A_r can be represented by our binary operators as follows.

$$A \cap A_1 = A_1 \quad \text{and} \quad A \cap A_r = A_r$$

$$A_1 \cup A_r = A$$

$$A - A_1 = A_r \quad \text{and} \quad A - A_r = A_1$$

This can be extended for more general cases. Suppose that a sparse array A has n dense subarrays (say $A_1, A_2, \cdots, A_n$) and the remaining part A_r. Then we have

$$A \cap A_i = A_i \quad \text{and} \quad A \cap A_r = A_r$$

$$(\cup_i^n A_i) \cup A_r = A$$

$$A - A_i = (\cup_{j \neq i}^n A_j) \cup A_r$$

. These operators can be used to estimate the probability of sparse ratio for arrays operated under F90 elementwise operations. We will use an example below to explain the key idea of our algorithm. Consider the example illustrated in Figure 8, arrays A and B have two subarrays A_1, A_2 and B_1, B_2, respectively. When they are applied to add function, we can calculate the probability for array C by calculating the sub-blocks of array C as follows.

$$C = \cup_i^n C_i$$

$$C_1 = A_1 \cap B_1$$

$$C_2 = A_1 \cap B_2$$

$$C_3 = A_2 \cap B_1$$

$$C_4 = A_2 \cap B_2$$

By using these operators, we can find the overlapping regions of dense subarrays in sparse arrays used in intrinsic functions. If the overlapping regions include several rectangular blocks, it can be represented as the union of several smaller rectangular regions. Thus, to estimate the sparsity ratio of the target array applied to an elemental array function, the first step is to find all overalpping and non-overlapping regions based on the index ranges of dense subarrays in input sparse arrays. The calculus for rectangular domains can be used to find the sparsity information for the target arrays.

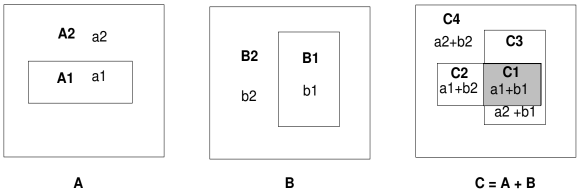

Figure 8. The overlapping and non-overlapping regions in two input sparse arrays under `addition` array operations.

2.3. Transformational Functions

For the case of transformational array operations with non-uniform distributions, we abstractly interpret the probability of an element to be zero pointwise for each target array element, and then use the sampling (extraction) algorithm developed earlier in Section 2.1 to obtain the sparsity structures. The sampling technique obtains regions of higher sparse density into blocks to gather regularity.

Note that we are working on a static compilation scheme, so the time spent with influence schemes did not give the overhead time for program executions. In our scheme, the sparsity of source arrays are profiled, and then sparsity structures of the rest of the arrays are obtained through our influence schemes. One will hope to employ our scheme into runtime to give runtime optimizations. In this case, we suggest a two-thread execution/compiler model proposed in[15] to be used to incorporate our influence scheme. The separation of the target code and the runtime compiler into separate modules can lead to overlap the execution time of these two modules in separate threads and have clear interfaces in the design of the compiler module.

3. Expremental Results

In this section, we report preliminary experimental results for applying our probabilistic inference schemes for sparsity ratio. First, we apply our sampling algorithm to test the effects of our scheme for non-uniform cases. Recall that the sampling technique is used to extract a set of blocks of sub-arrays with given density from the non-uniform sparse arrays. Once the sub-block of arrays are gathered, they can be used for further inference for the probability of sparse ratio of target arrays operated under F90 array intrinsics. This test set consists of benchmark matrice taken from the Harwell-Boeing sparse matrix collection, which is a set of standard test matrices arising from a wide range of sources and disciplines of real applications, such as problems in linear system, least squares, dynamic analysis, structural engineering, and astrophysics[4]. Table 1 gives the statistics for applying our algorithms on a large set of Harwell-Boeing sparse matrix collection. Among test set, the CIRPHYS dataset was used in circuit physics modeling, the ECONAUS dataset came from the discipline of economics modeling, the LANPRO dataset was for Lanczos with partial reorthogonalization, the LAPLACE dataset was used in partial differential equations at Boeing computer services, the LAPU dataset was used to model finite-element calculations at Harwell Lab, the LNS dataset came from the discipline of fluid flow modeling, the LSQ dataset was use for surveying, and the NUCL dataset was the applications in nuclear reactor modeling.

Table 1 illustrates our sampling results. In our experiment, the α and β specified in the Step 2 of the extraction algorithm are fixed at 0.03. Next, we extract the dense subarrays in these benchmarks in this test set with respect to 4×4 and 8×8 shape constraints. Table 1 is for these two shapes, respectively. The meanings of fields in these tables are described below.

- `shape` stands for the shape of the benchmark.
- `nnz` is the number of nonzero elements in the benchmark.
- `no. of dense subarrays` is the number of dense subarrays in the benchmark extracted by our algorithm.
- `average area` is the average of the product of extents in the shape of extracted dense subarrays.
- `nnz in dense subarray` is the summation of the number of nonzero elements contained in all extracted dense subarrays.
- `cover ratio` is the ratio of nnz in dense subarray to nnz.

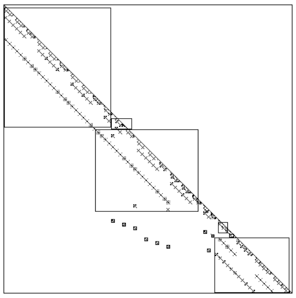

Figure 9. The extraction result for nos5.rsa.

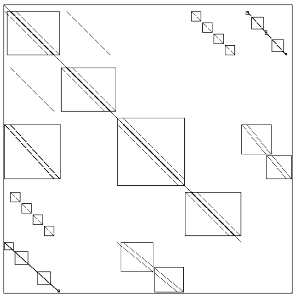

Figure 10. The extraction result for lns_511.rua.

Note that we are particularly interested in the ratio we cover. As the exact covering problem is NP-complete, our heuristic algorithm only gathers key dense blocks of the problem. Most of the dataset are covering more than 80% of the dataset. In the cases that the covering ratio is low, and we find the dataset is with diagonal structure or very thin-line of sparse structures. For example, the covering ratio for lns_131.rua is low, and it falls into this category. For these cases, we suggest a pattern-match style algorithm to use to match these special cases ahead of our sampling techniques to help this problem.

Fig. 9 and Fig. 10 give two examples to show the sampling results of our extraction algorithm with the initial shape constraint 4×4. It's for nos5.rsa and

benchmark	shape	nnz	initial shape constraint 4x4				initial shape constraint 8x8			
			no. of dense subarray	average cover area	nnz in dense subarrays	cover ratio	no. of dense subarray	average cover area	nnz in dense subarrays	cover ratio
cirphys.data	(991,991)	6027	126	1053.55	4342	0.720425	91	1584.41	4532	0.75195
econaus/mahindas.rua	(1258,1258)	7682	99	741.263	6662	0.867222	93	1287.31	7277	0.947279
econaus/orani678.rua	(2529,2529)	90158	150	4826.57	88336	0.979791	128	6491.17	87545	0.971018
lanpro/nos4.rsa	(100,100)	347	1	9120	335	0.965418	3	509	170	0.489914
lanpro/nos5.rsa	(468,468)	2820	16	4293.69	2632	0.933333	6	8676.17	2141	0.75922
lanpro/nos6.rsa	(675,675)	1965	10	5113.9	1592	0.810178	8	4541.38	1103	0.561323
lanpro/nos7.rsa	(729,729)	2673	27	2821.19	2402	0.898616	42	541.667	827	0.30939
laplace.data	(900,900)	4322	6	21670	3961	0.916474	6	21689	3962	0.916705
lapu.data	(25,16)	64	1	400	64	1	1	400	64	1
lns/lns_131.rua	(131,131)	536	20	55.95	235	0.438433	7	367.429	197	0.367537
lns/lns_3937.rua	(3937,3937)	25407	223	1441.69	11520	0.453418	271	2225.32	21028	0.827646
lns/lns_511.rua	(511,511)	2796	26	2395.88	2491	0.890916	35	1871.34	2534	0.906295
lns/lnsp_131.rua	(131,131)	536	6	1337.17	500	0.932836	2	6989.5	505	0.942164
lns/lnsp_511.rua	(511,511)	2796	25	3454.52	2703	0.966738	17	4832.59	2598	0.929185
lns/lnsp3937.rua	(3937,3937)	25407	375	1610.58	22737	0.894911	110	6149.15	20868	0.821348
lsq/illc1033.rra	(1033,320)	4732	96	1358.23	4517	0.954565	60	2276.65	4561	0.963863
lsq/illc1850.rra	(1850,712)	8758	229	812.821	8426	0.962092	124	1812.51	8323	0.950331
nucl/nnc1374.rua	(1374,1374)	8606	464	63.0345	7109	0.826052	28	7155.46	7284	0.846386
nucl/nnc261.rua	(261,261)	1500	86	75.3953	1305	0.87	3	10705	1211	0.807333
nucl/nnc666.rua	(666,666)	4044	226	69.5708	3532	0.873393	11	7035	3062	0.757171

Table 1. The statistics result of our extraction algorithm applied to Harwell-Boeing collection for initial shape constraint 4×4 and 8×8.

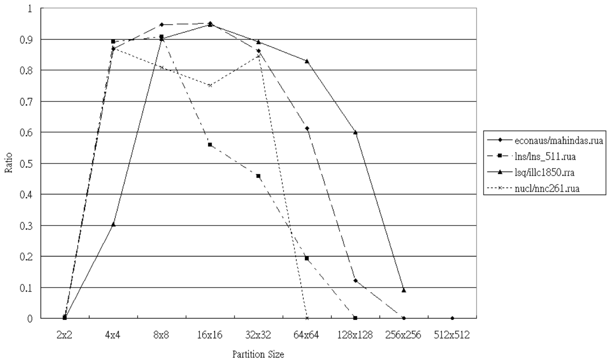

Figure 11. Cover ratios against initial shape constraint.

lns_511.rua applications among the Harwell-Boeing collection. The blocks are covered well and useful information are collected for sparsity structures. We can also see the merging effects of applying our algorithm in merging smaller blocks into larger blocks.

Focus in now directed to Figure 11 which compares the cover ratio for four different initial shapes. We can see that the cover ratios are better, when the initial shape constraint are 4×4 and 8×8. If the initial shape constraint is too large, it is difficult to extract suitable dense subarrays. In the practice, we can sample some points in the problem space to get the better results as shown in the figure.

Finally, we illustrate our probabilistic inference schemes for an OLAP application[8]. We use a 3D group-by operations related to OLAP (on-line analytical processing)[8, 9] for the example. The code segment in Figure 12 is a sequence of sum functions operated on sparse arrays ABC, AB, AC, BC, A, B, and C. In group-by operation of OLAP, the relationships among these sparse arrays can be represented as a a lattice. These are mainly dealing with large scale sparse data set. Again, our cost model for sparse matrix computations can be very useful to optimize the performances of the programs. Our work can help the code segment in Figure 12 for parallel sparse computations for the following ways. First, original Fortran 90 dense array intrinsics in the code can be are replaced with the corresponding sparse routines from our library. Second, our selection algorithm can be used to select the compression and distribution schemes for the algorithm. Finally, due to the natural of functionalities of group-by and OLAP to be represented as a lattice, if our compiler can recognize the computation to be a group-by operation, we can actually re-arrange the computation as well as the compression and distribution scheme for the group-by operation. This will reply on the precisions of the cost model and sparsity estimations.

In our experiment, we have the initial sparse array as a three-dimensional sparse array with the shape $10 \times 100 \times 1000$ and sparsity ratio 0.01. In this test case, nonzero elements are put in the $(5, i, j)$ positions, where $1 \leq i \leq 100$ and j is a multiplier of 10. Assuming that we have no knowledges of the distribution of sparse structures. Our sampling and extraction algorithm can be used to find the sparse structures. In this case, our extraction algorithm uses the shape constraint and sparsity constraint and $1 \times 50 \times 1$ and 0.05, respectively. In the first step of our extraction algorithm, the sparse array is partitioned into many

```
integer, dimension(1000,100,10) :: ABC
integer, dimension(1000,100) :: AB
integer, dimension(1000,10) :: AC
integer, dimension(100,10) :: BC
integer, dimension(1000) :: A
integer, dimension(100) :: B
integer, dimension(10) ::  C
integer :: ALL

AB = SUM(ABC,dim=3)
AC = SUM(ABC,dim=2)
BC = SUM(ABC,dim=1)
A  = SUM(AB,dim=2)
B  = SUM(AB,dim=1)
C  = SUM(AC,dim=1)
ALL = SUM(A)
```

Figure 12. Code segment of group-by operator.

$1 \times 50 \times 1$ subarrays. The dark subarrays shown in Figure 13 are kept. Next, each subarray will check its neighbors for merging. Therefore, two neighboring $1 \times 50 \times 1$ dense subarrays are merged into a larger $1 \times 100 \times 1$ dense subarray. Finally, nonzero elements can be reported as the original settings in the $(5, i, j)$ positions, where $1 \leq i \leq 100$ and j is a multiplier of 10. These sampling information for sparse arrays can then be used for cost modelling for compression schemes and distribution schemes.

Now we conduct experiments to demonstrate the performances of a 3D group-by operations related to OLAP on the above test set. In this experiment, a group-by operator is applied by performing a sequence of sum functions operated on sparse arrays ABC, AB, AC, BC, A, B, and C. Figure 14 shows two possible ways to achieve group-by operation for the same functionality. The RHS one is better than the LHS one for this sparse dataset, due to smaller number of computations as well as communications. The estimation is possible with the help of sparsity information estimated according to our proposed schemes. The reason that RHS is better than the one in LHS is explained as follows. After the first round of reduction, we will obtain the result for AB, AC, and BC. The sparsity for AB, AC, and BC can be estimated by the influence scheme provided in this work. AB is of size 1000×100 and with density 0.1, AC is of size 1000×10 and with density 0.01, and BC is of size 100×10 and with density 0.1 Detailed sampling of our scheme will find that the sparsity shape for AB is (i, j), $0 < i < 100$ and j is a multiplier of 10. The sparsity shape for AC is $(5, j)$, where j is a multiplier of 10. The sparsity shape for BC is $(5, i)$, where $0 < i < 100$. At this stage, there are two ways to produce A. One is done by reducing AB to A and the other is done by reducing AC to A. As AB is of size 1000×100 and with density 0.1, there are 10000 elements in the array. For array AC, there are only 100 elements needed for computations. Therefore, reducing AC to A in RHS is a better choice than reducing AB to A in LHS. Similarly, reducing AB to B in LHS requires 10000 elements of arrays, while reducing BC to B in RHS only requires 100 elements. Therefore, reducing BC to B is chosen in RHS for a better solution. Finally, reducing A to a scalar element in LHS requires 100 elements, while reducing C to a scalar element in RHS one element. Therefore, reducing C to one element is chosen in RHS for a better solution.

Table 2 show the performance results for these two assignments. Our selected suite is 1.33 times faster than the original code fragments with sparse intrinsics on 8-node IBM SP-2. The executions are done with our parallel sparse supports for Fortran 90 array intrinsics[14].

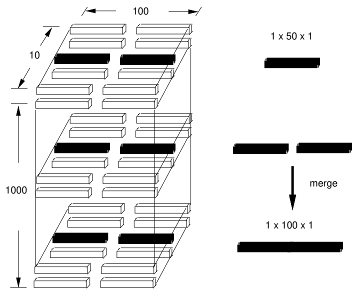

Figure 13. 3D example for OLAP.

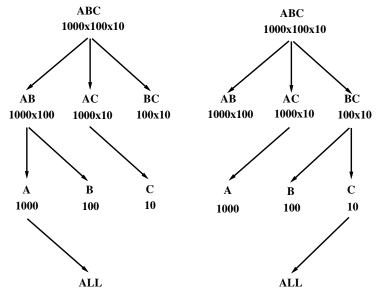

Figure 14. Two different ways to calculate the group-by operation according to the lattice.

no. of proc.	LHS Tree of Lattice	RHS Tree of Lattice
1	0.317	0.271
2	0.171	0.146
4	0.118	0.093
8	0.095	0.071

Table 2. Performance results of 3d group-by.

4. Related Work

Previous research work[1, 12, 13, 16] considered the distribution and compressed schemes related to one dimension and two dimension arrays. The work in [12] pioneered the research directions to annotate sparse notations and distributions for HPF-style languages. In our recent work, we built an extension applicable to higher-dimensional arrays. The approach to deal with higher-dimensional arrays is to employ 1-d and 2-d sparse arrays as the bases and compose them for higher-dimension arrays. The work in [2] addressed the sparse compiler issues. They also present a sampling scheme to find the sparse patterns. However, they do not have inference schemes for target arrays based on array operations.

The work in [3] also employs probablistic schemes to predict performances of direct mapping caches for sparse arrays. Their scheme is more a flavor of assuming uniform sparse distributions. We provide additional inference schemes for arrays operated basd on array operations and give sampling techniques for non-uniform cases. The work in [10] gives a generic abstraction (C++ template) for sparse data structure representations. A layer of abstractions for sparse structures is useful for building applications while avoiding details of low-level architectures.

5. Conclusion

In this paper, we provide probabilistic inference schemes to estimate sparsity ratio of target arrays operated based on Fortran 90 array operations and intrinsics. Our work advances the state of the art technology in estimating the sparsity ratio for arrays operated under Fortran 90 array operations and with non-uniform sparsity structures. These techniques are potentially very useful to guide compilers to improve the performances of sparse programs on advanced architectures.

References

[1] Bandera, G. and Zapata, E.L., *Sparse matrix block-cyclic redistribution; Parallel and Distributed Processing*, Proceedings of the IPPS/SPDP, Pages 355 - 359, 1999.

[2] A. J. C. Bik and H. A. G. Wijshoff, *Automatic data structure selection and transformation for sparse matrix computations*, IEEE Transactions on Parallel and Distributed Systems, 7 (1996), pp. 109–126.

[3] Doallo, R.; Fraguela, B.B.; Zapata, E.L., *Direct mapped cache performance modeling for sparse matrix operations*, Parallel and Distributed Processing, 1999. PDP '99.

[4] Iain S. Duff, Roger G. Grimes, and John G Lewis, *Sparse Matrix Test Problems*, ACM Trans. Math. Softw. pp.1-14, 15, 1, Mar. 1989.

[5] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.*

[6] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in MATLAB: Design and implementation*, SIAM J. on Matrix Anal. and Appl., 13 (1992), pp. 333—356.

[7] Y. Saad, *SPARSKIT: A basic tool kit for sparse computations, VERSION 2*, Computer Science Department, University of Minnesota, June 1994.

[8] Sanjay Goil and Alok Choudhary. High Performance OLAP and Data Mining on Parallel Computers, In Proceedings of IPPS/SPDP '98, Orlando, April, 1998.

[9] Mahesh V. Joshi, George Karypis, and Vipin Kumar. ScalParC: A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets, In Proceedings of IPPS/SPDP '98, Orlando, April, 1998.

[10] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar, *Next-generation generic programming and its application to sparse matrix computations*, Proceedings of the International conference on Supercomputing, 2000, Pages 88 - 99.

[11] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in Fortran 90: The Art of* Parallel *Scientific Computing.* Cambridge University Press, 1996.

[12] M. Ujaldon, E. Zapata, B. M. Chapman, and H. P. Zima, *New data–parallel language features for sparse matrix computations*, in Proc. 9th Int'l Parallel Processing Symp., Apr. 1995, pp. 742–749.

[13] Tyng–Ruey Chuang, Rong–Guey Chang, and Jenq Kuen Lee. Sampling and analytical techniques for data distribution of parallel sparse computation. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing.* Minneapolis, Minnesota, USA, March 1997. 8 pages. SIAM Press.

[14] Rong–Guey Chang, Tyng–Ruey Chuang, and Jenq Kuen Lee. Compiler Optimizations for Parallel Sparse Programs with Array Intrinsics of Fortran 90, International Conference on Parallel Processing (ICPP '99), IEEE Computer Society Press, September, 1999.

[15] Michael P. Plezbert and Ron K. Cytron. Does "just in time" = "better late then never". In *24th Annual SIGPLAN–SIGACT Symposium on Principle of Programming Languages.* Paris, France, 1997. 12 pages. ACM Press.

[16] Manuel Ujaldon and Emilio Zapata. Efficient resolution of sparse indirections in data–parallel compilers. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 117–126. Barcelona, Spain, July 1985. ACM Press.